# 字符集的比较

# GB2312

## 来历

**GB 2312** 或 **GB 2312–80** 是中华人民共和国国家标准简体中文字符集，全称《**信息交换用汉字编码字符集·基本集**》，又称**GBO**，由中国国家标准总局发布，1981年5月1日实施。GB 2312编码通行于中国大陆；新加坡等地也采用此编码。中国大陆几乎所有的中文系统和国际化的软件都支持GB 2312。[[1]](#footnote-0)为了用来表示汉字而发明。

## 分区表示

GB 2312中对所收汉字进行了“分区”处理，每区含有94个汉字／符号。这种表示方式也称为区位码。

01–09区为特殊符号。

16–55区为一级汉字，按拼音排序。

56–87区为二级汉字，按部首／笔画排序。

举例来说，“啊”字是GB 2312之中的第一个汉字，它的区位码就是1601。

10–15区及88–94区则未有编码。但在附录3，则在第10区推荐作为 GB 1988–80 中的94个图形字符区域（即第3区字符之半形版本）。[[2]](#footnote-1)

## 结构方法

在使用GB 2312的程序通常采用EUC储存方法，以便兼容于ASCII。这种格式称为EUC-CN。浏览器编码表上的“GB2312”就是指这种表示法。每个汉字及符号以两个字节来表示。第一个字节称为“高位字节”，第二个字节称为“低位字节”。“高位字节”使用了0xA1–0xF7（把01–87区的区号加上0xA0），“低位字节”使用了0xA1–0xFE（把01–94加上0xA0）。 由于一级汉字从16区起始，汉字区的“高位字节”的范围是0xB0–0xF7，“低位字节”的范围是0xA1–0xFE，占用的码位是72\*94=6768。其中有5个空位是D7FA–D7FE。例如“啊”字在大多数程序中，会以两个字节，0xB0（第一个字节）0xA1（第二个字节）储存。（与区位码对比：0xB0=0xA0+16,0xA1=0xA0+1）。[[3]](#footnote-2)

## 优点

适用于简体中文环境，属于中国国家标准，通行于大陆，新加坡等地也使用此编码[[4]](#footnote-3)

## 缺点

不兼容繁体中文，其汉字集合过少[[5]](#footnote-4)

# BIG5

## 来历

Big5，又称为大五码或五大码，是使用繁体中文（正体中文）社区中最常用的电脑汉字字符集标准，共收录13,060个汉字。中文码分为内码及交换码两类，Big5属中文内码，知名的中文交换码有CCCII、CNS11643。[[6]](#footnote-5)

“大五码”（Big5）是由台湾财团法人信息产业策进会为五大中文套装软件所设计的中文共通内码，在1983年12月完成公告[2][3]，隔年3月，信息产业策进会与台湾13家厂商签定“16位个人电脑套装软件合作开发（BIG-5）项目（五大中文套装软件）”[4]，因为此中文内码是为台湾自行制作开发之“五大中文套装软件”所设计的，所以就称为Big5中文内码[5][6][7][8]。五大中文套装软件虽然并没有如预期的取代国外的套装软件，但随着采用Big5码的国乔中文系统及倚天中文系统先后在台湾市场获得成功，使得Big5码深远地影响繁体中文电脑内码，直至今日。“五大码”的英文名称“Big5”后来被人按英文字序译回中文，以致现在有“五大码”和“大五码”两个中文名称。[[7]](#footnote-6)

Big5码的产生，是因为当时个人电脑没有共通的内码，导致厂商推出的中文应用软件无法推广，并且与IBM 5550、王安码等内码，彼此不能兼容；另一方面，台湾当时尚未推出中文编码标准。在这样的时空背景下，为了使台湾早日进入信息时代，所采行的一个项目；同时，这个项目对于以台湾为核心的亚洲繁体汉字圈也产生了久远的影响。[[8]](#footnote-7)

Big5产生前，研发中文电脑的朱邦复认为内码字集应该广纳所有的正异体字，以顾及如户政等应用上的需要，故在当时的内码会议中，建议希望采用他的五万多字的字库。工程师认为虽其技术可行，但是三个字节（超过两个字节）长度的内码却会造成英文显示屏画面映射成中文画面会发生文字无法对齐的问题，因为当时盛行之倚天中文系统画面系以两个字节文字宽度映射成一个中文字图样，英文软件中只要以两个英文字宽度去显示一个中文字，画面就不会乱掉，造成中文系统业者偏爱二个字节长度的内码[9]；此外以仓颉输入码压缩成的内码不具排序等功能，因此未被采用。1983年有人诬指朱邦复为共产党，其研究成果更不可能获采用。[10][[9]](#footnote-8)

在Big5码诞生后，大部分台湾的电脑软件都使用了Big5码，加上后来倚天中文系统的高度普及，使后来的微软Windows 3.x等亦予以采用。虽然后来台湾还有各种想要取代Big5码，像是倚天中文系统所推行的倚天码、台北市电脑公会所推动的公会码等，但是由于Big5字码已沿用多年，因此在习惯不易改变的情况下，始终无法成为主流字码。而台湾后来发展的国家标准CNS 11643中文标准交换码由于非一般的内码系统，是以交换使用为目的，受先天所限，必须使用至少三个字节来表示一个汉字，所以普及率远远不及Big5码。[[10]](#footnote-9)

在1990年代初期，当中国大陆的电子邮件和转码软件还未普遍之时，在深圳的港商和台商公司亦曾经使用Big5系统，以方便与总部的文件交流、以及避免为大陆的办公室再写一套不同内码的系统。使用简体中文的社区，最常用的是GB 2312、GBK及其后续的国标码（GB 18030）。[[11]](#footnote-10)

除了台湾外，其他使用繁体汉字的地区，如香港（香港增补字符集）、澳门，及使用繁体汉字的海外华人，都曾普遍使用Big5码做为中文内码及交换码。[[12]](#footnote-11)

## 结构方法

Big5码是一套双字节字符集，使用了双八码存储方法，以两个字节来安放一个字。第一个字节称为“高位字节”，第二个字节称为“低位字节”。“高位字节”使用了0x81-0xFE，“低位字节”使用了0x40-0x7E，及0xA1-0xFE。[[13]](#footnote-12)

## 优点

支持繁体中文

## 缺点

冲码问题，易造成乱码

# Utf-8/16

## 来历

1992年初，为创建良好的字节串编码系统以供多字节字符集使用，开始了一个正式的研究。ISO/IEC 10646的初稿中有一个非必须的附录，名为UTF。当中包含了一个供32位元的字符使用的字节串编码系统。这个编码方式的性能并不令人满意，但它提出了将0-127的范围保留给ASCII以兼容旧系统的概念。[[14]](#footnote-13)

1992年7月，X/Open委员会XoJIG开始寻求一个较佳的编码系统。Unix系统实验室（USL）的Dave Prosser为此提出了一个编码系统的建议。它具备可更快速实现的特性，并引入一项新的改进。其中，7位元的ASCII符号只代表原来的意思，所有多字节序列则会包含第8位元的符号，也就是所谓的最高有效位元。[[15]](#footnote-14)

1992年8月，这个建议由IBMX/Open的代表流传到一些感兴趣的团体。与此同时，贝尔实验室九号项目操作系统工作小组的肯·汤普逊对这编码系统作出重大的修改，让编码可以自我同步，使得不必从字符串的开首读取，也能找出字符间的分界。1992年9月2日，肯·汤普逊和罗勃·派克一起在美国新泽西州一架餐车的餐桌垫上描绘出此设计的要点。接下来的日子，Pike及汤普逊将它实现，并将这编码系统完全应用在九号项目当中，及后他将有关成果回馈X/Open。[[16]](#footnote-15)

1993年1月25-29日的在圣地牙哥举行的USENIX会议首次正式介绍UTF-8。[[17]](#footnote-16)

自1996年起，微软的CAB（MS Cabinet）规格在UTF-8标准正式落实前就明确容许在任何地方使用UTF-8编码系统。但有关的编码器实际上从来没有实现这方面的规格。[[18]](#footnote-17)

## 结构特点

UTF-8是UNICODE的一种变长度的编码表达方式《一般UNICODE为双位元组（指UCS2）》，它由肯·汤普逊（Ken Thompson）于1992年创建，现在已经标准化为RFC 3629。UTF-8就是以8位为单元对UCS进行编码，而UTF-8不使用大尾序和小尾序的形式，每个使用UTF-8存储的字符，除了第一个字节外，其余字节的头两个位元都是以"10"开始，使文字处理器能够较快地找出每个字符的开始位置。[[19]](#footnote-18)

但为了与以前的ASCII码兼容（ASCII为一个字节），因此UTF-8选择了使用可变长度字节来存储Unicode：[[20]](#footnote-19)

在ASCII码的范围，用一个字节表示，超出ASCII码的范围就用字节表示，这就形成了我们上面看到的UTF-8的表示方法，这様的好处是当UNICODE文件中只有ASCII码时，存储的文件都为一个字节，所以就是普通的ASCII文件无异，读取的时候也是如此，所以能与以前的ASCII文件兼容。[[21]](#footnote-20)

大于ASCII码的，就会由上面的第一字节的前几位表示该unicode字符的长度，比如110xxxxx前三位的二进制表示告诉我们这是个2BYTE的UNICODE字符；1110xxxx是个三位的UNICODE字符，依此类推；xxx的位置由字符编码数的二进制表示的位填入。越靠右的x具有越少的特殊意义。只用最短的那个足够表达一个字符编码数的多字节串。注意在多字节串中，第一个字节的开头"1"的数目就是整个串中字节的数目。[[22]](#footnote-21)

ASCII字母继续使用1字节存储，重音文字、希腊字母或西里尔字母等使用2字节来存储，而常用的汉字就要使用3字节。辅助平面字符则使用4字节。[[23]](#footnote-22)

在UTF-8文件的开首，很多时都放置一个U+FEFF字符（UTF-8以EF,BB,BF代表），以显示这个文本文件是以UTF-8编码。[[24]](#footnote-23)

## 优点

ASCII是UTF-8的一个子集。因为一个纯ASCII字符串也是一个合法的UTF-8字符串，所以现存的ASCII文本不需要转换。为传统的扩展ASCII字符集设计的软件通常可以不经修改或很少修改就能与UTF-8一起使用。[[25]](#footnote-24)

使用标准的面向字节的排序例程对UTF-8排序将产生与基于Unicode代码点排序相同的结果。（尽管这只有有限的有用性，因为在任何特定语言或文化下都不太可能有仍可接受的文字排列顺序。）[[26]](#footnote-25)

UTF-8和UTF-16都是可扩展标记语言文档的标准编码。所有其它编码都必须通过显式或文本声明来指定。[2][[27]](#footnote-26)

任何面向字节的字符串搜索算法都可以用于UTF-8的数据（只要输入仅由完整的UTF-8字符组成）。但是，对于包含字符记数的正则表达式或其它结构必须小心。[[28]](#footnote-27)

UTF-8字符串可以由一个简单的算法可靠地识别出来。就是，一个字符串在任何其它编码中表现为合法的UTF-8的可能性很低，并随字符串长度增长而减小。举例说，字符值C0,C1,F5至FF从来没有出现。为了更好的可靠性，可以使用正则表达式来统计非法过长和替代值（可以查看W3 FAQ: Multilingual Forms上的验证UTF-8字符串的正则表达式）。[[29]](#footnote-28)

与UCS-2的比较：ASCII转换成UCS-2，在编码前插入一个0x0。用这些编码，会含括一些控制符，比如"或 '/'，这在UNIX和一些C函数中，将会产生严重错误。因此可以肯定，UCS-2不适合作为Unicode的外部编码，也因此诞生了UTF-8。[[30]](#footnote-29)

## 缺点

编写不良的解析器

一份写得很差（并且与当前标准的版本不兼容）的UTF-8解析器可能会接受一些不同的伪UTF-8表示并将它们转换到相同的Unicode输出上。这为设计用于处理八位表示的校验例程提供了一种遗漏信息的方式。[[31]](#footnote-30)

不利于正则表达式检索

正则表达式可以进行很多英文高级的模糊检索。例如，[a-h]表示a到h间所有字母。同样GBK编码的中文也可以这样利用正则表达式，比如在只知道一个字的读音而不知道怎么写的情况下，也可用正则表达式检索，因为GBK编码是按读音排序的。只是UTF-8不是按读音排序的，所以会对正则表达式检索造成不利影响。但是这种使用方式并未考虑中文中的破音字，因此影响不大。Unicode是按部首排序的，因此在只知道一个字的部首而不知道如何发音的情况下，UTF-8可用正则表达式检索而GBK不行。[[32]](#footnote-31)

其他

与其他Unicode编码相比，特别是UTF-16，在UTF-8中ASCII字符占用的空间只有一半，可是在一些字符的UTF-8编码占用的空间就要多出1/3，特别是中文、日文和韩文（CJK）这样的方块文字。[[33]](#footnote-32)

# Unicode

## 来历

Unicode是为了解决传统的字符编码方案的局限而产生的，例如ISO 8859-1所定义的字符虽然在不同的国家中广泛地使用，可是在不同国家间却经常出现不兼容的情况。很多传统的编码方式都有一个共同的问题，即容许电脑处理双语环境（通常使用拉丁字母以及其本地语言），但却无法同时支持多语言环境（指可同时处理多种语言混合的情况）。[[34]](#footnote-33)

Unicode编码包含了不同写法的字，如“ɑ／a”、“強／强”、“戶／户／戸”。然而在汉字方面引起了一字多形的认定争议（详见中日韩统一表意文字主题）。[[35]](#footnote-34)

在文字处理方面，统一码为每一个字符而非字形定义唯一的代码（即一个整数）。换句话说，统一码以一种抽象的方式（即数字）来处理字符，并将视觉上的演绎工作（例如字体大小、外观形状、字体形态、文体等）留给其他软件来处理，例如网页浏览器或是文字处理器。[[36]](#footnote-35)

目前，几乎所有电脑系统都支持基本拉丁字母，并各自支持不同的其他编码方式。Unicode为了和它们相互兼容，其首256字符保留给ISO 8859-1所定义的字符，使既有的西欧语系文字的转换不需特别考量；并且把大量相同的字符重复编到不同的字符码中去，使得旧有纷杂的编码方式得以和Unicode编码间互相直接转换，而不会丢失任何信息。举例来说，全角格式区块包含了主要的拉丁字母的全角格式，在中文、日文、以及韩文字形当中，这些字符以全角的方式来呈现，而不以常见的半角形式显示，这对竖排文字和等宽排列文字有重要作用。[[37]](#footnote-36)

在表示一个Unicode的字符时，通常会用“U+”然后紧接着一组十六进制的数字来表示这一个字符。在基本多文种平面（英文：Basic Multilingual Plane，简写BMP。又称为“零号平面”、plane 0）里的所有字符，要用四个数字（即两个char,16bit ,例如U+4AE0，共支持六万多个字符）；在零号平面以外的字符则需要使用五个或六个数字。旧版的Unicode标准使用相近的标记方法，但却有些微小差异：在Unicode 3.0里使用“U-”然后紧接着八个数字，而“U+”则必须随后紧接着四个数字。[[38]](#footnote-37)

## 结构特点

统一码的编码方式与ISO 10646的通用字符集概念相对应。目前实际应用的统一码版本对应于UCS-2，使用16位的编码空间。也就是每个字符占用2个字节。这样理论上一共最多可以表示216（即65536）个字符。基本满足各种语言的使用。实际上当前版本的统一码并未完全使用这16位编码，而是保留了大量空间以作为特殊使用或将来扩展。[[39]](#footnote-38)

上述16位统一码字符构成基本多文种平面。最新（但未实际广泛使用）的统一码版本定义了16个辅助平面，两者合起来至少需要占据21位的编码空间，比3字节略少。但事实上辅助平面字符仍然占用4字节编码空间，与UCS-4保持一致。未来版本会扩充到ISO 10646-1实现级别3，即涵盖UCS-4的所有字符。UCS-4是一个更大的尚未填充完全的31位字符集，加上恒为0的首位，共需占据32位，即4字节。理论上最多能表示231个字符，完全可以涵盖一切语言所用的符号。[[40]](#footnote-39)

基本多文种平面的字符的编码为U+hhhh，其中每个h代表一个十六进制数字，与UCS-2编码完全相同。而其对应的4字节UCS-4编码后两个字节一致，前两个字节则所有位均为0。[[41]](#footnote-40)

关于统一码和ISO 10646及UCS的详细关系，见通用字符集。[[42]](#footnote-41)

Unicode的实现方式不同于编码方式。一个字符的Unicode编码是确定的。但是在实际传输过程中，由于不同系统平台的设计不一定一致，以及出于节省空间的目的，对Unicode编码的实现方式有所不同。Unicode的实现方式称为Unicode转换格式（Unicode Transformation Format，简称为UTF）[[43]](#footnote-42)

例如，如果一个仅包含基本7位ASCII字符的Unicode文件，如果每个字符都使用2字节的原Unicode编码传输，其第一字节的8位始终为0。这就造成了比较大的浪费。对于这种情况，可以使用UTF-8编码，这是一种变长编码，它将基本7位ASCII字符仍用7位编码表示，占用一个字节（首位补0）。而遇到与其他Unicode字符混合的情况，将按一定算法转换，每个字符使用1-3个字节编码，并利用首位为0或1进行识别。这样对以7位ASCII字符为主的西文文档就大幅节省了编码长度（具体方案参见UTF-8）。类似的，对未来会出现的需要4个字节的辅助平面字符和其他UCS-4扩充字符，2字节编码的UTF-16也需要通过一定的算法进行转换。[[44]](#footnote-43)

再如，如果直接使用与Unicode编码一致（仅限于BMP字符）的UTF-16编码，由于每个字符占用了两个字节，在麦金塔电脑（Mac）机和个人电脑上，对字节顺序的理解是不一致的。这时同一字节流可能会被解释为不同内容，如某字符为十六进制编码4E59，按两个字节拆分为4E和59，在Mac上读取时是从低字节开始，那么在Mac OS会认为此4E59编码为594E，找到的字符为“奎”，而在Windows上从高字节开始读取，则编码为U+4E59的字符为“乙”。就是说在Windows下以UTF-16编码保存一个字符“乙”，在Mac OS环境下打开会显示成“奎”。此类情况说明UTF-16的编码顺序若不加以人为定义就可能发生混淆，于是在UTF-16编码实现方式中使用了大端序（Big-Endian，简写为UTF-16 BE）、小端序（Little-Endian，简写为UTF-16 LE）的概念，以及可附加的字节顺序记号解决方案，目前在PC机上的Windows系统和Linux系统对于UTF-16编码默认使用UTF-16 LE。（具体方案参见UTF-16）[[45]](#footnote-44)

此外Unicode的实现方式还包括UTF-7、Punycode、CESU-8、SCSU、UTF-32、GB18030等，这些实现方式有些仅在一定的国家和地区使用，有些则属于未来的规划方式。目前通用的实现方式是UTF-16小端序（LE）、UTF-16大端序（BE）和UTF-8。在微软公司Windows XP附带的记事本（Notepad）中，“另存为”对话框可以选择的四种编码方式除去非Unicode编码的ANSI（对于英文系统即ASCII编码，中文系统则为GB2312或Big5编码）外，其余三种为“Unicode”（对应UTF-16 LE）、“Unicode big endian”（对应UTF-16 BE）和“UTF-8”。[[46]](#footnote-45)

目前辅助平面的工作主要集中在第二和第三平面的中日韩统一表意文字中，因此包括GBK、GB18030、Big5等简体中文、繁体中文、日文、韩文以及越南喃字的各种编码与Unicode的协调性被重点关注。考虑到Unicode最终要涵盖所有的字符。从某种意义而言，这些编码方式也可视作Unicode的出现于其之前的既成事实的实现方式，如同ASCII及其扩展Latin-1一样，后两者的字符在16位Unicode编码空间中的编码第一字节各位全为0，第二字节编码与原编码完全一致。但上述东亚语言编码与Unicode编码的对应关系要复杂得多。[[47]](#footnote-46)

## 优点

它能容纳全世界所有的语言文字，主要用来解决使用不同语言编写文本，以及字符集种类繁多所带来的各种问题。Unicode为每个字符提供一个唯一标识的二进制编码，能够满足跨语言，跨平台进行文本转换和处理的要求。任何其他编码的字符都可以无损地转换为Unicode。[[48]](#footnote-47)

## 缺点

空间浪费严重

# 各编码集之间的转换

## Big5/UTF-8编码转换的模组

方式一[[49]](#footnote-48)：

use Encode::compat;

use Encode qw(from\_to);

my $string = "中文";

#Big5 转UTF-8

from\_to($string, 'big5', 'utf8');

print "$string\n";

#UTF-8 转Big5

from\_to($string, 'utf8', 'big5');

print "$string\n";

方式二[[50]](#footnote-49)：

use Encode::compat;

use Encode qw(encode decode);

my $string = "中文";

#Big5 转UTF-8

my $utf8 = encode('utf8', decode('big5', $string));

print "$utf8\n";

my $big5 = encode('big5', decode('utf8', $utf8));

print "$big5\n";

## 字符集转换: Unicode - Ansi[[51]](#footnote-50)

[cpp] view plain copy

string CXX::UnicodeToAnsi ( const wstring& wstrSrc )

{

/\*!< 分配目标空间, 一个16位Unicode字符最多可以转为4个字节\*/

//int iAllocSize = static\_cast<int>( wstrSrc.size() \* 4 + 10 );

int iAllocSize = WideCharToMultiByte(CP\_THREAD\_ACP,0,wstrSrc.c\_str(),-1,NULL,0,NULL,NULL);

char\* pwszBuffer = new char[ (UINT)iAllocSize ];

memset(pwszBuffer,0,iAllocSize);

if ( NULL == pwszBuffer )

{

return "";

}

int iCharsRet = WideCharToMultiByte( CP\_THREAD\_ACP, 0, wstrSrc.c\_str(), -1,pwszBuffer, iAllocSize, NULL, NULL );

/\*!< 成功 \*/

string strRet;

if ( 0 < iCharsRet )

{

(void)strRet.assign ( pwszBuffer, static\_cast<size\_t>( iCharsRet ) );

}

/\*!< 释放内存 \*/

delete[] pwszBuffer;

return strRet;

}

## 字符集转换: Ansi - Unicode[[52]](#footnote-51)

[cpp] view plain copy

wstring CXX::AnsiToUnicode (const string& strSrc )

{

/\*!< 分配目标空间 \*/

int iAllocSize = MultiByteToWideChar(CP\_ACP,0,strSrc.c\_str(),-1,NULL,NULL);

WCHAR\* pwszBuffer = new WCHAR[ (UINT)iAllocSize ];

if ( NULL == pwszBuffer )

{

return L"";

}

int iCharsRet = MultiByteToWideChar( CP\_ACP, 0, strSrc.c\_str(),-1,pwszBuffer, iAllocSize );

/\*!< 成功 \*/

wstring wstrRet;

if ( 0 < iCharsRet )

{

(void)wstrRet.assign ( pwszBuffer, static\_cast<size\_t>( iCharsRet ) );

}

/\*!< 释放内存 \*/

delete[] pwszBuffer;

return wstrRet;

}

## 字符集转换: Ansi - UTF8[[53]](#footnote-52)

[cpp] view plain copy

string CFunction::AnsiToUtf8 ( const string& strSrc )

{

/\*!< 分配目标空间, 长度为 Ansi 编码的两倍 \*/

int iAllocSize = MultiByteToWideChar(CP\_ACP,0,strSrc.c\_str(),-1,NULL,NULL);

WCHAR\* pwszBuffer = new WCHAR[ (UINT)iAllocSize ];

if ( NULL == pwszBuffer )

{

return "";

}

int iCharsRet = MultiByteToWideChar( CP\_ACP, 0, strSrc.c\_str(), -1, pwszBuffer, iAllocSize );

/\*!< 成功 \*/

wstring wstrTemp;

if ( 0 < iCharsRet )

{

(void)wstrTemp.assign ( pwszBuffer, static\_cast<size\_t>( iCharsRet ) );

}

/\*!< 释放内存 \*/

delete[] pwszBuffer;

return UnicodeToUtf8( wstrTemp );

}

## 字符集转换: UTF8 - Ansi[[54]](#footnote-53)

[cpp] view plain copy

string CFunction::Utf8ToAnsi( const string& strSrc )

{

wstring wstrTemp = Utf8ToUnicode ( strSrc );

int iAllocSize = WideCharToMultiByte(CP\_ACP,0,wstrTemp.c\_str(),-1,NULL,0,NULL,NULL);

char\* pszBuffer = new char[ (UINT)iAllocSize ];

if ( NULL == pszBuffer )

{

return "";

}

int iCharsRet = WideCharToMultiByte( CP\_ACP, 0, wstrTemp.c\_str(),-1,pszBuffer, iAllocSize, NULL, NULL );

string strRet;

if ( 0 < iCharsRet )

{

(void)strRet.assign(pszBuffer, static\_cast<size\_t>(iCharsRet));

}

delete[] pszBuffer;

return strRet;

}

## 字符集转换: Unicode - UTF8

[cpp] view plain copy

string CFunction::UnicodeToUtf8 ( const wstring& wstrSrc )

{

/\*!< 分配目标空间, 一个16位Unicode字符最多可以转为4个字节 \*/

int iAllocSize = WideCharToMultiByte(CP\_UTF8,0,wstrSrc.c\_str(),-1,NULL,0,NULL,NULL);

char\* pszBuffer = new char[ (UINT)iAllocSize ];

if ( NULL == pszBuffer )

{

return "";

}

int iCharsRet = WideCharToMultiByte( CP\_UTF8, 0, wstrSrc.c\_str(), -1, pszBuffer, iAllocSize, NULL, NULL );

/\*!< 成功 \*/

string strRet;

if ( 0 < iCharsRet )

{

(void)strRet.assign ( pszBuffer, static\_cast<size\_t>( iCharsRet ) );

}

/\*!< 释放内存 \*/

delete[] pszBuffer;

return strRet;

}

## 字符集转换: UTF8 - Unicode[[55]](#footnote-54)

[cpp] view plain copy

wstring CFunction::Utf8ToUnicode ( const string& strSrc )

{

int iAllocSize = MultiByteToWideChar(CP\_UTF8, 0, strSrc.c\_str(),-1,NULL,NULL);

WCHAR\* pwszBuffer = new WCHAR[ (UINT)iAllocSize ];

if ( NULL == pwszBuffer )

{

return L"";

}

int iCharsRet = MultiByteToWideChar( CP\_UTF8, 0, strSrc.c\_str(), -1, pwszBuffer, iAllocSize );

wstring wstrRet;

if ( 0 < iCharsRet )

{

(void)wstrRet.assign(pwszBuffer, static\_cast<size\_t>(iCharsRet));

}

delete[] pwszBuffer;

return wstrRet;

}

## 判断是否是UTF8编码[[56]](#footnote-55)

[cpp] view plain copy

bool CFunction::IsUTF8(const void\* pBuffer, long size)

{

bool IsUTF8 = true;

unsigned char\* start = (unsigned char\*)pBuffer;

unsigned char\* end = (unsigned char\*)pBuffer + size;

while (start < end)

{

if (\*start < 0x80) // (10000000): 值小于0x80的为ASCII字符

{

start++;

}

else if (\*start < (0xC0)) // (11000000): 值介于0x80与0xC0之间的为无效UTF-8字符

{

IsUTF8 = false;

break;

}

else if (\*start < (0xE0)) // (11100000): 此范围内为2字节UTF-8字符

{

if (start >= end - 1)

break;

if ((start[1] & (0xC0)) != 0x80)

{

IsUTF8 = false;

break;

}

start += 2;

}

else if (\*start < (0xF0)) // (11110000): 此范围内为3字节UTF-8字符

{

if (start >= end - 2)

break;

if ((start[1] & (0xC0)) != 0x80 || (start[2] & (0xC0)) != 0x80)

{

IsUTF8 = false;

break;

}

start += 3;

}

else

{

IsUTF8 = false;

break;

}

}

return IsUTF8;

}

1. 来自GB2312维基百科 [↑](#footnote-ref-0)
2. 来自GB2312维基百科 [↑](#footnote-ref-1)
3. 来自GB2312维基百科 [↑](#footnote-ref-2)
4. 来自http://blog.csdn.net/xueshandugu/article/details/22106293 [↑](#footnote-ref-3)
5. 来自http://blog.csdn.net/xueshandugu/article/details/22106293 [↑](#footnote-ref-4)
6. 来自BIG5维基百科 [↑](#footnote-ref-5)
7. 来自BIG5维基百科 [↑](#footnote-ref-6)
8. 来自BIG5维基百科 [↑](#footnote-ref-7)
9. 来自BIG5维基百科 [↑](#footnote-ref-8)
10. 来自BIG5维基百科 [↑](#footnote-ref-9)
11. 来自BIG5维基百科 [↑](#footnote-ref-10)
12. 来自BIG5维基百科 [↑](#footnote-ref-11)
13. 来自BIG5维基百科 [↑](#footnote-ref-12)
14. 来自utf-8维基百科 [↑](#footnote-ref-13)
15. 来自utf-8维基百科 [↑](#footnote-ref-14)
16. 来自utf-8维基百科 [↑](#footnote-ref-15)
17. 来自utf-8维基百科 [↑](#footnote-ref-16)
18. 来自utf-8维基百科 [↑](#footnote-ref-17)
19. 来自utf-8维基百科 [↑](#footnote-ref-18)
20. 来自utf-8维基百科 [↑](#footnote-ref-19)
21. 来自utf-8维基百科 [↑](#footnote-ref-20)
22. 来自utf-8维基百科 [↑](#footnote-ref-21)
23. 来自utf-8维基百科 [↑](#footnote-ref-22)
24. 来自utf-8维基百科 [↑](#footnote-ref-23)
25. 来自utf-8维基百科 [↑](#footnote-ref-24)
26. 来自utf-8维基百科 [↑](#footnote-ref-25)
27. 来自utf-8维基百科 [↑](#footnote-ref-26)
28. 来自utf-8维基百科 [↑](#footnote-ref-27)
29. 来自utf-8维基百科 [↑](#footnote-ref-28)
30. 来自utf-8维基百科 [↑](#footnote-ref-29)
31. 来自utf-8维基百科 [↑](#footnote-ref-30)
32. 来自utf-8维基百科 [↑](#footnote-ref-31)
33. 来自utf-8维基百科 [↑](#footnote-ref-32)
34. 来自Unicode维基百科 [↑](#footnote-ref-33)
35. 来自Unicode维基百科 [↑](#footnote-ref-34)
36. 来自Unicode维基百科 [↑](#footnote-ref-35)
37. 来自Unicode维基百科 [↑](#footnote-ref-36)
38. 来自Unicode维基百科 [↑](#footnote-ref-37)
39. 来自Unicode维基百科 [↑](#footnote-ref-38)
40. 来自Unicode维基百科 [↑](#footnote-ref-39)
41. 来自Unicode维基百科 [↑](#footnote-ref-40)
42. 来自Unicode维基百科 [↑](#footnote-ref-41)
43. 来自Unicode维基百科 [↑](#footnote-ref-42)
44. 来自Unicode维基百科 [↑](#footnote-ref-43)
45. 来自Unicode维基百科 [↑](#footnote-ref-44)
46. 来自Unicode维基百科 [↑](#footnote-ref-45)
47. 来自Unicode维基百科 [↑](#footnote-ref-46)
48. 来自http://www.cqvip.com/read/read.aspx?id=71769074504849524853514952 [↑](#footnote-ref-47)
49. 来自https://translate.google.co.jp/translate?hl=zh-CN&sl=zh-TW&u=http://www.vixual.net/blog/archives/59&prev=search [↑](#footnote-ref-48)
50. 来自https://translate.google.co.jp/translate?hl=zh-CN&sl=zh-TW&u=http://www.vixual.net/blog/archives/59&prev=search [↑](#footnote-ref-49)
51. 来自清风似水流的专栏的博客包括以下的代码 [↑](#footnote-ref-50)
52. 来自清风似水流的专栏的博客包括以下的代码 [↑](#footnote-ref-51)
53. 来自清风似水流的专栏的博客包括以下的代码 [↑](#footnote-ref-52)
54. 来自清风似水流的专栏的博客包括以下的代码 [↑](#footnote-ref-53)
55. 来自清风似水流的专栏的博客包括以下的代码 [↑](#footnote-ref-54)
56. 来自清风似水流的专栏的博客包括以下的代码 [↑](#footnote-ref-55)